Datatypes in L2

Datatypes in L2

Nick Chapman[1], Simon Finn[1], Michael P. Fourman[2]

[1] Abstract Hardware Ltd.
[2] Abstract Hardware Ltd. and Edinburgh University

Abstract. We describe the axiomatisation of a subset of Standard ML's datatypes in L2 (the LAMBDA Logic). The subset includes parameterisation and mutual recursion but has restrictions on the use of function type construction. We sketch a set-theoretic model for these datatypes. Finally, we briefly discuss the relationship between L2's datatypes and datatypes in HOL.

1 Introduction

LAMBDA is a proof assistant designed for the specification and verification of digital systems. User-defined datatypes are an important tool for expressing well-structured specifications.

Early versions of LAMBDA (prior to LAMBDA 4.0) used a 'free' logic, allowing terms that may not denote. This logic could support a rich set of datatypes – essentially[3] the same as Standard ML [8]. The semantics of these datatypes can be described in a standard domain-theoretic way [4]; in fact the presence of the existence predicate, $\mathbf{E}$, means that the information-theoretic domain ordering is actually expressible in the logic (which therefore contains LCF as a sub-logic). In about 1991, we decided to change the logic used within the LAMBDA system. The basic reason for this change is that the old logic appears to be too expressive for the intended usage of the LAMBDA system; hardware designers are rarely impressed by having to consider the subtle distinction between the two functions $\lambda x.\ \bot$ and $\bot$, for example.

The new LAMBDA logic – now known as L2 – borrows heavily from HOL, but with a concrete syntax based on Standard ML. The philosophy of LAMBDA is somewhat different from that of HOL; rather than trying to reduce every proof to a small number of axioms, we are (relatively) happy to allow the system to construct new axioms from user-supplied definitions. This difference becomes most apparent in the treatment of recursive functions – where LAMBDA doesn't require function definitions to be primitive recursive (see [3] for details) – and in the current work on datatype definition.

SML-style datatype definitions provide a natural way to express specifications, as we had discovered using the 'old' logic, so we wanted to provide them

[3] Standard ML allows the definition of datatypes that are too general – in the sense that you can't traverse them with a well-typed recursive function; LAMBDA doesn't support these datatypes.

as part of L2 too. Melham [7] had already shown how to embed a useful subset of the datatype language within HOL. However, it is simply not possible to provide the full generality of SML datatypes within HOL's set-theoretic model – a simple argument about set cardinalities shows this; Gunter [5] provides a constructive proof – in HOL – that it's not possible in any other sort of model for HOL either.

Given the constraint of keeping the logic consistent, what kind of datatypes can we allow? We believe that the version[4] of L2 supported by LAMBDA 4.3 (as described in [2]), which includes parameterisation, mutual recursion and the (limited) use of function space constructors, is pretty close to the maximal datatype language that can be supported by a HOL-like logic. The L2 datatype sublanguage is, in fact, very similar to the 'full class of [datatype] specifications' outlined by Gunter in [6]. The principal difference is that L2 datatype definitions are able to make use of existing type constructors (and we give sufficient conditions for this use to be 'legal') whereas Gunter excludes this, although she adds:

> 'It is also possible to extend the notion of admissibility to include occurrences of certain kinds of type constructors, but the precise definition of this case is quite complicated and we omit it here.'

2 Design Aims

Our design aims for datatypes in L2 are:

1. The syntax should be the same as that used for datatypes in Standard ML.
2. The class of datatypes provided should be as rich as possible within L2's classical, polymorphic, higher-order type-theory.
3. Any restrictions imposed on the ML datatypes should be semantically rather than syntactically based.
4. The induction rules generated by the system should be easy to use within the LAMBDA proof system.

We have made the following restrictions with respect to Standard ML's datatypes:

2.1 Function Space Restriction

Every datatype must be small enough to be modelled as a set. In particular, within the body of a datatype definition, there must be no occurrence of that datatype on the left-hand side of a function arrow. This restriction is treated semantically, so that

[4] L2 evolves as our ideas evolve; in particular, the original version of L2 – supported by LAMBDA 4.0 – had much poorer support for datatypes.

```
datatype ('a,'b) arrow = Arrow of 'a -> 'b;
datatype bad = Bad of (bad,bool) arrow;
```

is, of course, illegal.

To enforce the semantic restriction, LAMBDA computes, for each datatype and each type parameter, whether that parameter is 'dirty' (occurs on the left of a function-space arrow or as a subtype parameter) or 'clean'. (Subtype parameters are 'dirty' because L2 subtype construction is not, in general, monotonic; increasing the size of the carrier of the parameter to a polymorphic subtype may *decrease* the size of the carrier of the result. In fact – with a suitable subtype predicate – the size of the subtype can be arbitrarily related to the size of the parameter type.) Recursive instances of the datatype within the body of its definition are legal only if they occur in clean positions.

For simplicity, we make the conservative assumption that a parameterised datatype actually depends on all of its type parameters. This means that LAMBDA may occasionally reject definitions which we could, semantically, allow. For example:

```
datatype 'a ignore = X
datatype funny = Y of funny ignore -> bool
```

LAMBDA will reject this definition of `funny`, because it assumes that `funny ignore` – which occurs on the left of a function arrow – actually depends on `funny`. If this restriction became irksome, we could keep track of which datatypes embed which of their parameters but, for the moment, this seems an unnecessary refinement.

2.2 Non-emptiness Restriction

Every datatype must be non-empty. For example, the definition

```
datatype empty = Empty of empty;
```

is *not* allowed. Note that we impose a semantic restriction rather than saying something syntactic like 'every datatype must contain a nullary constructor'. This means that we can allow useful definitions such as

```
datatype 'a gentree = Tree of 'a gentree list * 'a;
```

LAMBDA enforces the non-emptiness constraint on datatypes by means of an abstract interpretation. Each L2 parameterised datatype is associated with a boolean function; this function has one boolean parameter for each type parameter of the datatype and returns a boolean result. For a non-parameterised datatype, this function degenerates into a single boolean value.

Informally, the interpretation of the boolean value `true` is that we know that the carrier of the corresponding type is non-empty. (As in HOL, all *legal* L2 types have non-empty carriers. Since we are trying to *establish* that a given datatype is

legal, however, we can't make that assumption here. As we will see later, empty sets *do* make an appearance in our model for datatypes; what we have to show is that all *legal* types are modelled by non-empty sets.) The boolean function corresponding to an L2 datatype tells us whether we can construct an element of that datatype, on the assumption that we are given elements of some of the parameter types (those for which the parameter in the abstraction is **true**).

A recursive datatype definition will give rise to a recursive equation for the corresponding boolean function. We solve such recursive equations by taking the least fixed point of the corresponding functional i.e. we assume that the datatype is empty unless we can prove otherwise. (We can only guarantee to find a fixed point because we know that the functional is *monotonic*. This wouldn't necessarily be the case if we allowed the recursively defined datatype to occur on the left-hand side of a function arrow. In practice, this means that we have to check that this doesn't occur *before* we check for non-emptiness.) The datatype definition is legal (or at least, not illegal on the grounds of emptiness) if the boolean function returns **true** when all its parameters are **true**. An example may make this clearer. Suppose we have the L2 definitions:

```
datatype ('a,'b) choice = A of 'a | B of 'b;
datatype 'a list = nil | :: of 'a * 'a list;
datatype 'a tree = ('a, 'a tree list) choice;
```

The corresponding boolean functions would satisfy the following equations:

```
f_choice(a,b) = a \/ b
f_list(a)     = true \/ (a /\ f_list(a))
f_tree(a)     = f_choice(a,f_list(f_tree(a)))
```

which have the least-fixed point solutions:

```
f_choice(a,b) = a \/ b
f_list(a)     = true
f_tree(a)     = true
```

2.3 Parameter Uniformity Restriction

For a parameterised datatype, all instances of the datatype occurring in the body of the declaration must have identical parameters to the defining instance. For example, the following definition is *not* allowed:

```
datatype 'a up = Up of 'a | Down of ('a up) up;
```

This restriction is needed to ensure that the induction rule generated for the datatype (see below) is well-typed.[5]

[5] Such datatypes, although legal in Standard ML, are actually useless in practice for just the same reason – the impossibility of writing well-typed recursive functions to traverse them.

For mutually recursive datatypes, we have an obvious[6] generalisation of this restriction. All the datatypes being defined together must have the same number of type parameters and all instances of any of the mutually recursive datatypes occurring in the body of any of the declarations must have the same type parameters as occur in the head of that declaration. For example,

```
datatype 'a gentree     = Tree of 'a gentreelist * 'a
     and 'b gentreelist = List of 'b gentree list
```

is legal, but

```
datatype ('a,'b) swap1 = X | Y of ('a,'b) swap2
     and ('c,'d) swap2 = A | B of ('d,'c) swap1
```

is not, because the occurrence of `('d,'c)swap1` within the definition of `('c,'d)swap2` is illegal – the type parameters don't occur in the same order.

3 Axiomatisation

Given a legal L2 datatype definition, LAMBDA produces a number of rules to axiomatise the properties of that datatype. These rules fall into 3 classes:

1. For each unary constructor, LAMBDA produces a rule stating that it is a 1-1 function i.e. two terms built using the constructor are equal only if they have equal arguments.
2. For each pair of distinct constructors, LAMBDA produces a rule stating that two terms built using different constructors are unequal.
3. For each datatype, LAMBDA produces an induction rule stating that every value in the datatype can be built using one of the constructors.

The first two classes of rules are uninteresting and will not be discussed further. By contrast, constructing appropriate induction rules is somewhat non-trivial and – for parameterised or mutually-recursive datatypes – also requires the axiomatisation of a number of auxiliary functions, as will be described below.

The first of these auxiliary functions is the **extend** function. The **extend** function corresponding to a parameterised datatype takes one parameter – a predicate – for each type parameter of the datatype definition and produces a predicate which operates on the datatype itself. Roughly speaking, the **extend** function applies each predicate to all subterms of the corresponding type and conjoins the results. For example, the L2 datatype definition

```
datatype 'a tree =
   Empty
 | Just of 'a
 | Pair of bool -> 'a
 | Many of 'a tree * 'a tree list;
```

[6] This generalisation is so 'obvious', in fact, that we needed 6 months to discover it.

would produce the induction rule:

```
G // H |- forall t,l. Ptree#(t)
                    /\ extend'list (fn x => Ptree#(x)) l
                          ->> Ptree#(Many (t,l))
G // H |- forall f. Ptree#(Pair f)
G // H |- forall x. Ptree#(Just x)
G // H |- Ptree#(Empty)
------------------------------------------------------
G // H |- forall t. Ptree#(t)
```

which uses the function **extend'list** – previously generated from the definition of the **list** datatype – and would also define the **extend'tree** function:

```
fun extend'tree p Empty        = TRUE
  | extend'tree p (Just x)     = p x
  | extend'tree p (Pair f)     = forall b:bool. p (f b)
  | extend'tree p (Many(t,l)) =
      extend'tree p t /\ extend'list (extend'tree p) l
```

so that **tree** can itself be used in future datatype definitions. In addition to the explicit induction rules, LAMBDA allows the definition of 'primitive recursive' functions that manipulate the newly introduced datatype. For example, LAMBDA would recognise the following function definitions as primitive recursive:

```
fun countItems Empty     = 0
  | countItems (Just x) = 1
  | countItems (Pair f) = 2
  | countItems (Many (t,tl)) =
      countItems t + countItemsInList tl

and countItemsInList []       = 0
  | countItemsInList (t::ts) =
      countItems t + countItemsInList ts
```

We discuss LAMBDA's definition of 'primitive recursive' in more detail later. The combination of the explicit datatype axioms together with the principle of definition of primitive recursive functions is categorical i.e. they determine the structure of the values of the datatype (up to isomorphism).

When we have mutually recursive datatype definitions, expressing the induction rules requires an extra family of auxiliary functions – the **convert** functions. For example the definition

```
datatype 'a T  = Node of 'a * 'a TL
     and 'b TL = Nil | Cons of 'b T * 'b TL
```

generates the following pair of induction rules:

```
G // H |- forall x,tl. convert'T'TL (fn t => PT#(t)) tl
                          ->> PT#(Node(x,tl))
-------------------------------------------------------
G // H |- forall t. PT#(t)

G // H |- forall t,tl. convert'TL'T (fn tl => PTL#(tl)) t
                    /\ PTL#(tl)
                         ->> PTL#(Cons(t,tl))
G // H |- PTL#(Nil)
-------------------------------------------------------
G // H |- forall tl. PTL#(tl)
```

Each of these induction rules uses an additional '**convert**' auxiliary function. The intuition behind the **convert** functions is that the predicate **convert'X'Y P** holds of an object **y** of type **Y** precisely if **P** holds of all the immediate subterms of **y** which are of type **X** e.g. **convert'T'TL** converts an (inductive) predicate on **T** into a predicate on **TL**. The definition of these functions is

```
fun convert'T'TL f Nil          = TRUE
  | convert'T'TL f (Cons (x,y)) = f x /\ convert'T'TL f y

fun convert'TL'T f (Node (x,y)) = f y
```

In general, defining n mutually recursive datatypes generates n induction rules and n groups of **convert** functions, where each group contains $n-1$ mutually recursive functions.

It would have been possible to define the induction rules *without* introducing the auxiliary **convert** functions. For example, we could have produced the following, apparently simpler, rules:

```
G // H |- forall x,tl. PTL#(tl) ->> PT#(Node(x,tl))
G // H |- forall t,tl. P#(t) /\ PTL#(tl) ->> PTL#(Cons(t,tl))
G // H |- PTL#(Nil)
-------------------------------------------------------
G // H |- forall t. PT#(t)

G // H |- forall x,tl. PTL#(tl) ->> PT#(Node (x,tl))
G // H |- forall t,tl. P#(t) /\ PTL#(tl) ->> PTL#(Cons(t,tl))
G // H |- PTL#(Nil)
-------------------------------------------------------
G // H |- forall tl. PTL#(tl)
```

This – allowing for differences in the logic – is how we treated mutually recursive datatypes in LAMBDA 3.X. The reason that we don't use these seductively simple rules within the current version of LAMBDA is that they are *hard to use*. There are two reasons for this:

1. If we are using an induction rule to perform case analysis rather than full-blown induction, the 'simple' rules force us to consider constructors from all the mutually-recursive datatypes, rather than only the datatype of interest.
2. When we use an induction rule for 'real' induction, we need to instantiate the meta-variables (PT and PTL above) to produce the concrete induction scheme for the particular predicate that we wish to prove. LAMBDA's higher-order unification will instantiate one of these meta-variables for us when we apply the induction rule, but we will then have to instantiate the other(s) by hand. What makes this particularly annoying is that we normally need to define some auxiliary functions in order to perform the instantiation – we need, in fact, to define the **convert** functions by hand.

The apparently more complex induction rules than LAMBDA now generates solve both of these pragmatic problems.

4 Primitive Recursion

LAMBDA will recognise a function definition as primitive recursive if it can show by a simple syntactic check that the corresponding function always is total.[7] The syntactic conditions that a primitive-recursive function must fulfill are as follows. Suppose the function is defined by a series of clauses, each with the function symbol applied to n symbols. For each occurrence of the function symbol in the body of any clause

1. The function must be applied to at least 1 argument.
2. For some i, $0 \leq i < n$, the first i arguments must be identical to the first i patterns at the head of that clause. The $i+1$'th argument must be strictly smaller than the $i+1$'th pattern.

For mutually recursive functions, we slightly generalise the above rules. Suppose several mutually-recursive functions are defined by clauses. Then, for each occurrence of any of the mutually-recursive functions in the body of any of the clauses:

1. The function must be applied to at least 1 argument.
2. For some i, $0 \leq i < n$, where n is the number of patterns occurring in that particular clause[8], the first i arguments must be identical to the first i patterns. The $i+1$'th argument must be strictly smaller than the $i+1$'th pattern.

What does 'strictly smaller' mean? An expression is *smaller* than a pattern if one of the following holds:

[7] LAMBDA also allows the definition of non primitive-recursive functions. To make effective use of such a function the user has to discharge a side condition that says, essentially, that the function 'terminates'. This will be discussed in detail in [3].

[8] For mutually recursive functions, n may vary from clause to clause because different functions may have different numbers of parameters; for each individual function, the number of patterns in each clause should still be constant.

1. The pattern is a variable (N.B. *not* a constructor) and the expression is the same variable or consists of the application of that variable to one or more arguments.
2. The pattern is a nullary constructor and the expression is the same constructor.
3. The pattern and expression each consist of an application of the same unary constructor and the argument in the expression is smaller than the argument in the pattern.
4. The expression and pattern are both labelled records (this includes tuples) with the same labels and each subexpression is smaller than the corresponding subpattern.
5. The expression is smaller than a strict subpattern of the pattern.

An expression is *strictly smaller* than a pattern if it is smaller than the pattern, but not identical to it.

5 Axiomatisation within LAMBDA

In this section we describe the concrete form of the induction rules and auxiliary functions produced by LAMBDA.

5.1 Auxiliary Functions – `extend`

As noted above, LAMBDA generates higher-order '`extend`' functions which take one parameter – a predicate – for each type parameter of the original datatype definition and produce a predicate which operates on the datatype itself. We characterised this function as applying each predicate to all subterms of the corresponding type and conjoining the results. This characterisation of `extend` is slightly too simple:

1. If the type parameter is embedded in the range of a function type, then the `extend` function must quantify over the range of the function, as for `Pair` in the above example. This means that we are interpreting 'subterm' in a semantic rather than a syntactic sense.
2. If the type parameter is ever embedded in the domain of a function type – i.e. the type is 'dirty' – then the corresponding predicate is never applied. This doesn't cause a problem because we define `extend` functions precisely so that we can use parameterised datatypes in the definition of new, indirectly recursive, datatypes (as we used `list` in the definition of `tree`, for example) and our restriction on datatype definitions excludes recursion through such 'dirty' parameters.

In general, the mutually recursive datatype definition

datatype $(\text{'a}_{11}, \ldots, \text{'a}_{1n})\ D_1$ = ...
and ...
and $(\text{'a}_{k1}, \ldots, \text{'a}_{kn})\ D_k$ = ... | C_{ki} | ... | C_{kj} **of** t_{kj} | ...

gives rise to the k functions **extend'D**$_1$ ... **extend'D**$_k$. Conceptually, we define these functions as described below; in practice LAMBDA also performs a 'pattern-lifting' phase (essentially beta-reduction plus simplification of trivial conjuncts) to improve the readability of the generated definitions.
For nullary constructors, the **extend** function always returns **TRUE**

extend'D$_x$ p$_1$ ...p$_n$ C$_{xi}$ = **TRUE**

For unary constructors, its value depends on the structure of the type of the constructor

extend'D$_x$ p$_1$ ...p$_n$ (C$_{xi}$ v$_{xi}$) = [[t$_{xi}$]] v$_{xi}$

where the operation [[_]] is defined by

[[t]] = fn v => TRUE,
where t is any type containing no instance of a clean parameter.

[['a$_{xj}$]] = p$_j$,
where 'a$_{xj}$ is the j'th parameter type and 'a$_{xj}$ is a clean parameter.

[[('a$_{x1}$, ..., 'a$_{xn}$)D$_y$]] = **extend'D**$_y$ p$_1$ ...p$_n$,
where D$_y$ is one of the mutually recursive datatypes[9] – possibly D$_x$ itself.

[[(t$_1$, ..., t$_l$)D]] = extend'D [[t$_1$]] ...[[t$_l$]],
where D is some other datatype constructor and some t$_j$ contains a clean parameter. Note that this condition logically implies that the j'th parameter position of D must be clean.

[[{l$_j$: t$_j$}]] = fn {l$_j$: v$_j$} => $\bigwedge_j$([[t$_j$]] v$_j$),
where {l$_j$: t$_j$} is a labelled record type and some t$_j$ contains an instance of a clean parameter.

[[t$_1$ -> t$_2$]] = fn f => forall x : t$_1$. [[t$_2$]] (f x),
where t$_2$ contains an instance of a clean parameter.

The case [[(t$_1$, ..., t$_l$)T]] where T is a type abbreviation is handled by expanding the abbreviation.

Note that the predicate p$_j$ will never be applied if the corresponding type parameter, 'a$_j$, is dirty. We could eliminate these parameters altogether, but we choose not to do so; this means that if, in the future, we change the definition of 'clean' – to take account of datatypes which don't embed their arguments, for example – we won't have to change the type of any existing **extend** function.

[9] This rule means that the **extend** functions for mutually recursive datatypes must also be mutually recursive.

5.2 Auxiliary Functions – convert

As noted above, the predicate **convert'X'Y P** holds of an object **y** of type **Y** precisely if **P** holds of all the immediate subterms of **y** which are of type **X** i.e. **convert'X'Y** converts an (inductive) predicate on **X** into a predicate on **Y**. This means that the **convert'X'Y** function will have type

```
(X -> om) -> Y -> om
```

Suppose we have the mutually recursive datatype definition

```
datatype ('a11, ..., 'a1n) D1 = ...
and ...
and ('ak1, ..., 'akn) Dk = ...| Cki | ...| Ckj of tkj | ...
```

Then, for nullary constructors, the **convert** function always returns **TRUE**

```
convert'Dx'Dy Px Cyi = TRUE
```

For unary constructors, **convert** function depends on the structure of the type of the constructor

```
convert'Dx'Dy Px (Cyj vyj) = [[tyj]] vyj
```

where the compilation operation **[[_]]** is here defined to be

[[t]] = fn x => TRUE,
where **t** is any type containing no instance of any of the mutually-recursive datatypes.

[[('ay1, ..., 'ayn)Dx]] = Px

[[('ay1, ..., 'ayn)Dz]] = convert'Dx'Dz Px,
where D_z, distinct from D_x but possibly the same as D_y, is one of the mutually-recursive datatypes.

[[(t1, ..., tl)D]] = extend'D [[t1]] ... [[tl]],
where **D** is a previously-defined datatype constructor.

[[{lj : tj}]] = fn {lj : vj} => $\bigwedge_j$**([[tj]] vj),**
where $\{l_j : t_j\}$ is a labelled record type.

[[t1 -> t2]] = fn f => forall x : t1. [[t2]] (f x)

As for the **extend** functions, we handle the case **[[(t1, ..., tl)T]]** where **T** is a type abbreviation by expanding the abbreviation.

As for the **extend** family of functions, LAMBDA performs pattern-lifting to optimise the definitions produced by the above naive algorithm.

5.3 Induction Rules – Construction

Suppose we have the mutually recursive datatype definition

```
datatype ('a11, ..., 'a1n) D1 = ...
and ...
and ('ak1, ..., 'akn) Dk = ...| Cki | ...| Ckj of tkj | ...
```

LAMBDA will produce k induction rules, one for each datatype. The rule for each datatype consists of a conclusion plus one premiss for each constructor of that datatype. For the datatype $\mathtt{D}_x$, the conclusion will be

```
G // H |- forall w : ('ax1, ..., 'axn)Dx. PDx#(w)
```

The premiss corresponding to a nullary constructor, $\mathtt{C}_{xi}$, of type $\mathtt{D}_x$ will be

```
G // H |- PDx#(Cxi)
```

For a unary constructor, $\mathtt{C}_{xi}$, of type $\mathtt{t}_{xi}$ `->` `('a`$x1$`, ..., 'a`xn`)`$\mathtt{D}_x$, the premiss will be

```
G // H |- forall v̄xi. prexi ⟶ PDx#(patxi)
```

where $\langle\bar{\mathtt{v}}_{xi},\mathtt{pat}_{xi},\mathtt{pre}_{xi}\rangle$ = `[[`$\mathtt{t}_{xi}$`]]` and the compilation operator `[[_]]` is defined as follows:

`[[{`$\mathtt{l}_j$ `:` $\mathtt{t}_j$`}]]` = $\langle @_j\ \bar{\mathtt{v}}_j,\ \{\mathtt{l}_j : \mathtt{pat}_j\},\ \bigwedge_j \mathtt{pre}_j\rangle$,
where $\{\mathtt{l}_j : \mathtt{t}_j\}$ is a labelled record type, $\langle\bar{\mathtt{v}}_j, \mathtt{pre}_j, \mathtt{pat}_j\rangle$ = `[[`$\mathtt{t}_j$`]]`, and we use the notation '$@_j\ \bar{\mathtt{v}}_j$' to represent vector concatenation.

`[[('a`$x1$`, ..., 'a`xn`)`$\mathtt{D}_x$`]]` = $\langle$`v,v,`$\mathtt{PD}_x$`#(v)`$\rangle$,
where `v` is a new variable.

`[[('a`$x1$`, ..., 'a`xn`)`$\mathtt{D}_y$`]]` =
`<v,v,convert'`$\mathtt{D}_x$`'`$\mathtt{D}_y$ `(fn z =>` $\mathtt{PD}_x$`#(z)) v>`,
where $\mathtt{D}_y$ is another of the mutually-recursive datatypes and `v` is a new variable.

`[[(`$\mathtt{t}_1$, ..., $\mathtt{t}_l$`)D]]` =
`<v,v,extend'D (fn pat`$_1$ `=> pre`$_1$`) ... (fn pat`$_l$ `=> pre`$_l$`) v>`,
where `D` is a previously-defined datatype constructor, there is an occurrence of one of the mutually recursive datatypes in at least one of the $\mathtt{t}_j$, $\langle\bar{\mathtt{v}}_j, \mathtt{pat}_j, \mathtt{pre}_j\rangle$ = `[[`$\mathtt{t}_j$`]]`, and `v` is a new variable.

`[[`$\mathtt{t}_1$ `->` $\mathtt{t}_2$`]]` = `<f,f,forall x. pre[v ⟵ (f x)]>`,
where $\mathtt{t}_2$ contains one of the mutually recursive datatypes,
`<v, v, pre>` = `[[`$\mathtt{t}_2$`]]`, `v` is a variable, and `f` and `x` are new variables.

`[[`t_1 `->` t_2`]] =`
`<f,f,forall x. (fn pat => pre) (f x)>,`
where t_2 contains one of the mutually recursive datatypes,
`<`$\overline{\mathtt{v}}$`, pat, pre> = [[`t_2`]]`, `pat` is not a variable, and `f` and `x` are new variables.

As before, we handle the case `[[(`t_1`, ..., `t_l`)T]]` where `T` is a type abbreviation by expanding the abbreviation.

`[[t]] = <v,v,TRUE>`,
where v is a new variable and none of the above rules apply.

6 Sketch of Semantics

How do we build a set-theoretic model for L2 datatypes? In general the L2 model would be similar to Pitts' set-theoretic model for HOL [9]. We then have to explain how to add the denotations of recursive datatypes.[10] We then proceed in something like the following stages:

1. We model an L2 datatype as the least fixed point of a monotonic function on a suitable lattice of sets (with a suitable appeal to Tarski's Fixed Point Theorem justify the existence of a fixed point.) The restrictions that we have made on the form of L2 datatypes are just what we need to ensure that such a monotonic function exists and that the resulting fixed point is a *non-empty* set. In particular:
   (a) We made the restriction that all instances of the datatype occurring in the body of the declaration must the same parameters as the defining instance. This means that we can treat the parameter types as fixed when we construct the fixed point and then parameterise the result. (If we didn't have this restriction we would need to find the fixed point of a functional rather than just a function.)
   (b) The restriction that recursive occurrences of the datatype occurring in the body of its definition may only occur in 'clean' positions is precisely what we need to show that the function is monotonic. (Here we need to make the assumption that previously defined parameterised datatypes give rise to functions that are indeed monotonic in their 'clean' parameters. We can justify this by an induction on the number of previously-defined datatypes.)
2. We next need to show that the newly-defined parameterised datatype is a monotonic function of its 'clean' parameters. This should be standard argument involving the least fixed points of monotonic functions.

[10] We also have to explain how to handle non-primitive recursive functions; this will be treated in [3] – the techniques used there are remarkably similar to our treatment of datatypes.

3. At this stage in the argument, we have established that the L2 datatype can be represented as a set. We next have to show that the abstract interpretation is correct i.e. that it is conservative in its prediction about whether the datatype is non-empty.
4. Next we have to consider the **extend** functions. If we regard them as functions on sets (represented by their characteristic functions) we can see that we can define the **extend** function for a datatype – as a least fixed point – in much the same way as we defined the datatype itself.
5. Finally the induction rules can be justified by an argument involving least fixed points of monotonic functions. The only complication here is that **extend** functions appear to ignore their 'dirty' arguments i.e. P_i is treated as if it were `fn _ => true` whenever the `i`'th parameter type is dirty. This isn't actually a problem, because when the `i`'th parameter type is dirty, P_i actually *is* `fn _ => true` i.e. 'dirty' types are treated as fixed and non-empty throughout the proof. (We could simplify this proof by making the definition of the **extend** function match the datatype definition more exactly, but that wouldn't be very user-friendly.)

7 Relationship of L2 datatypes to HOL datatypes

As we noted in the introduction, the main technical difference between L2 datatypes and Gunter's[6] HOL datatypes is that L2 datatype definitions may make use of existing type constructors. In some respects, this difference is not important because it is always possible to expand out the use of such type constructors by introducing new, mutually-recursive, datatypes. For example, we could treat the definition:

```
datatype 'a gentree = Tree of 'a gentree list * 'a;
```

as if it were:

```
datatype 'a gentree  =
  Tree of 'a gentree_list * 'a

and 'a gentree_list =
  Nil | Cons of 'a gentree * 'a gentree_list
```

If we do this consistently, we can reduce a collection of L2 datatype definitions into a form equivalent to Gunter's[6] 'full class of specifications'. (Our function-space condition is sufficient to show that the expanded form meets Gunter's admissibility conditions.) This is perhaps the simplest way to give a meaning to L2 datatype definitions.

Doing this at the source level would have a distinct price however. The two types `'a gentree list` and `'a gentree_list` are isomorphic but they are not identical. This means that it would not be possible to apply useful general purpose functions such as `map` to an object of type `'a gentree_list` and so it would be necessary to develop a separate theory of lists for each such 'instantiation' of the `list` constructor.

8 Future Work

When we started the first draft of this paper, we believed that our characterisation of L2 datatypes was essentially complete, and that the datatypes we described were in some sense 'maximal' for a HOL-like logic.[11] Since then, we have had a couple of ideas for extensions.

We currently treat all subtyping as 'dirty'. This means, for example, that if we add an integer index to each node of a `gentree` and specify, using subtyping, that such indices must all be distinct then we can't use the resulting type in any future datatype definition. Given the current HOL (or L2) type scheme, this seems to be unavoidable. The problem is that we can't tell whether or not the subtype predicate makes the subtype non-monotonic in the size of the subtype's parameters, so we have regard *all* the subtype's parameters as potentially non-monotonic i.e. 'dirty'.

We believe that it may be possible to make a small change to the type scheme to remove this restriction, although we haven't worked out all the details yet. The basic idea is to borrow Standard ML's concept of 'imperative' type variables to keep track of which type parameters are 'clean' and which are 'dirty'. Standard functions have normal 'applicative' types, but (rather ironically) quantifiers get 'imperative' types rather like Standard ML's `ref` constructor.

Two reviewers pointed out the close relationship between the definition of a datatype and the associated principle of definition for functions on that datatype. Although we have successfully defined induction rules using parameterised datatypes, we have not done so well with the definitional principle. For example, we defined the function `countItems` as:

```
fun countItems Empty     = 0
  | countItems (Just x) = 1
  | countItems (Pair f) = 2
  | countItems (Many (t,tl)) =
      countItems t + countItemsInList tl

and countItemsInList []       = 0
  | countItemsInList (t::ts) =
      countItems t + countItemsInList ts
```

Here the recursion pattern for `countItems`, in particular the use of the auxiliary function `countItemsInList`, is exactly what one would expect if we had defined a local `'a tree_list` datatype rather than using `'a tree list` in the datatype definition. A more natural definition of `countItems` would be something like:

```
fun countItems Empty     = 0
  | countItems (Just x) = 1
```

[11] With the exception – already noted – that we can define a better function space restriction by keeping track of whether a type constructor actually uses all its type parameters.

```
| countItems (Pair f) = 2
| countItems (Many (t,tl)) =
    countItems t +
      fold'list (0, op +) (map'list countItems tl);
```

Here we are assuming that the `fold'list` and `map'list` functions would be automatically generated from the datatype definition for `list` and, crucially, that we can regard this definition as primitive recursive. There is clearly considerable scope for investigating LAMBDA's definition of 'primitive recursion'.

9 Acknowledgements

Our treatment builds on the work of Matt Fairtlough[1] and we are grateful to Matt for many useful discussions. We would also like the thank the anonymous referees many of whose constructive suggestions have been incorporated into this paper. The work reported in this paper was partially funded by the projects *Formal System Design* (IED/2/1292) and *Synthesis, Optimisation and Analysis* (JESSI AC8).

References

1. Matt Fairtlough, Research into ML Datatypes, in *Formal System Design (IED Project 1292) Deliverable D13*, Edinburgh University, February 1992.
2. Simon Finn, Michael P. Fourman, L2 – The LAMBDA Logic, in *LAMBDA 4.3 Reference Manuals*, Abstract Hardware Limited, September 1993.
3. Simon Finn, Michael Fourman, John Longley, *Partial Functions in a Total Setting*, in preparation.
4. M.P. Fourman and W.K. Phoa, A Proposed Categorical Semantics for Pure ML, in *ICALP '92 International Colloquium on Automata, Languages, and Programming, Wien Austria*, Springer-Verlag LNCS, 1993.
5. Elsa L. Gunter, Why We Can't have SML Style datatype Declarations in HOL, in *Higher Order Logic Theorem Proving and its Applications (HOL'92)*, ed. L.J.M. Claessen, M.J.C. Gordon, North-Holland 1993.
6. Elsa L. Gunter, A Broader Class of Trees for Recursive Type definitions in HOL, in *Higher Order Logic Theorem Proving and its Applications (HUG'93)*, ed. Jeffrey J. Joyce, Carl-Johan H. Seger, Lecture Notes in Computer Science 780, Springer-Verlag 1994.
7. Thomas F. Melham, Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic, PhD Thesis and Technical Report 201, University of Cambridge, August 1990.
8. Robin Milner, Mads Tofte and Robert Harper, *The Definition of Standard ML*, MIT Press, 1990.
9. Andy Pitts, Set-Theoretic Semantics, in *The HOL System DESCRIPTION*, HOL88 Documentation, 1991.